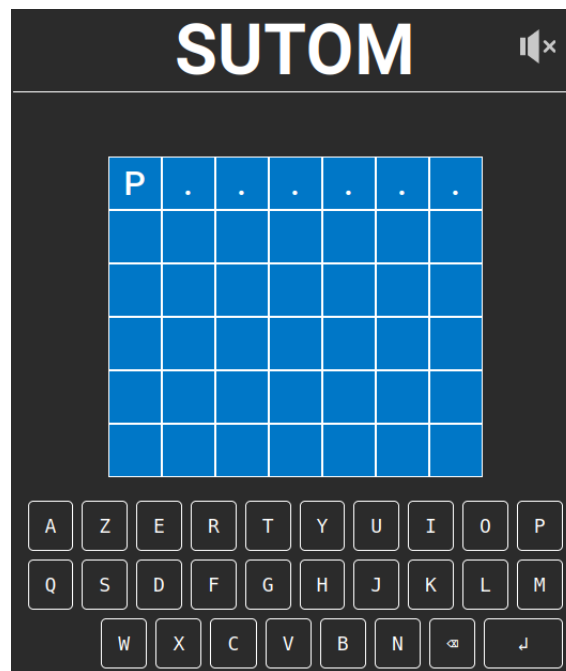


1 Préambule

Objectif :

L'objectif de ce projet est de réaliser un simulateur du jeu *sutom* version française du jeu *Wordle* lui-même inspiré du jeu télévisé français *motus*.

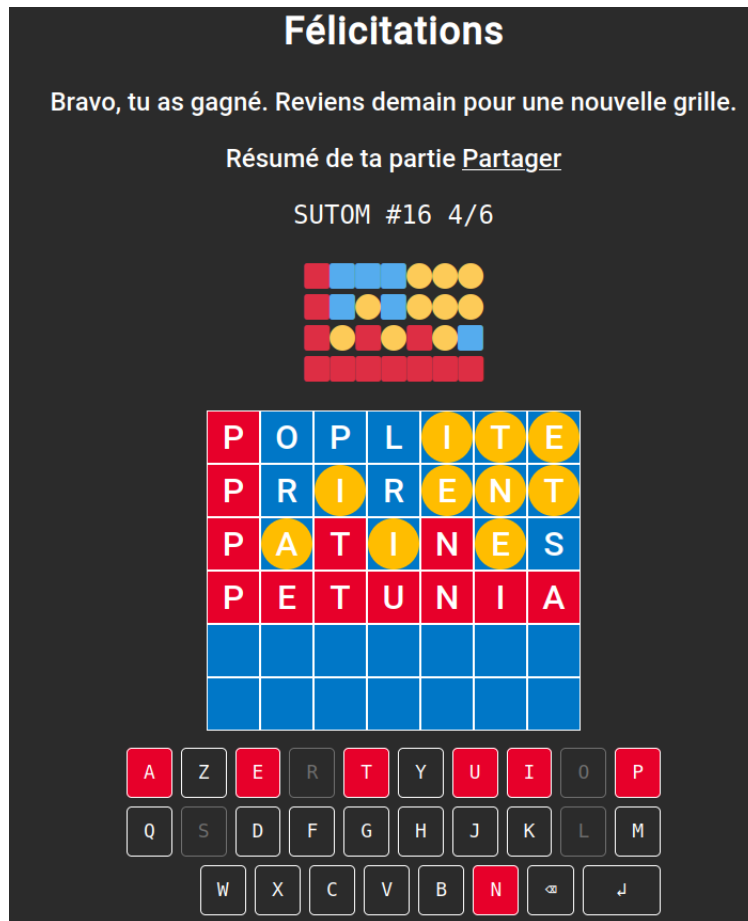
L'interface se présente ainsi :



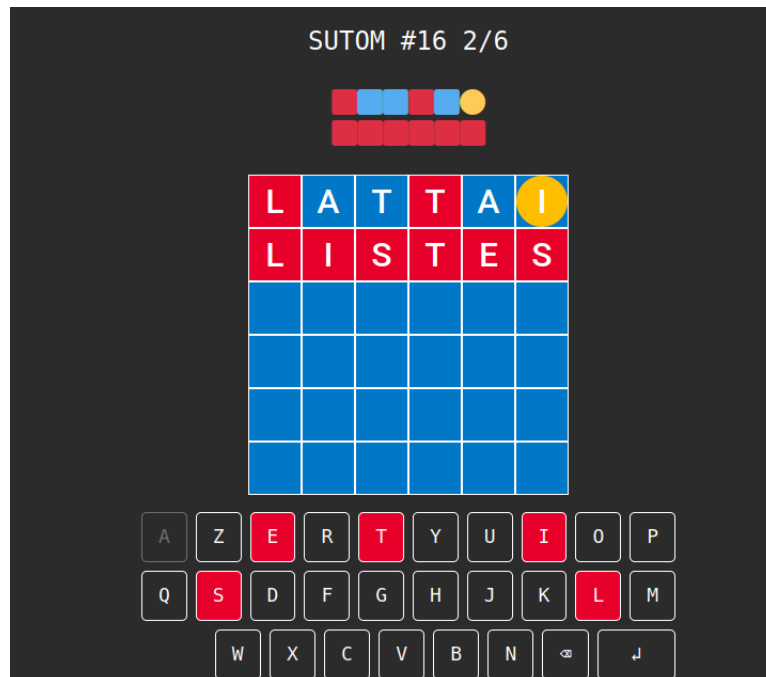
Les règles du jeu sont les suivantes :

- Vous avez six essais pour deviner le mot du jour.
- Vous ne pouvez proposer que des mots commençant par la même lettre que le mot recherché, et qui se trouvent dans notre dictionnaire.
- Les lettres entourées d'un carré rouge sont bien placées, les lettres entourées d'un cercle jaune sont mal placées (mais présentes dans le mot).
- Les lettres qui restent sur fond bleu ne sont pas dans le mot.
- Il y a un mot par jour, entre 6 et 9 lettres, et il est identique pour tout le monde.

Un exemple de recherche du mot du dimanche 23/01/2022 avec l'assistant que vous programmerez dans ce projet :



On donne un autre exemple avec la lettre 'T' qui n'apparaît qu'une seule fois dans le mot secret, on considère que le 'T' surnuméraire n'appartient pas au mot secret.



Dans ce projet vous travaillerez sur des interfaces simplifiées où :

- une lettre absente du mot secret sera représentée par le caractère '*'
- une lettre mal placée sera représentée par le caractère '+'

- les lettres cachées seront représentées par le caractère '?' dans le premier affichage du mot secret par l'ordinateur.


```
Saisir l'affichage de l'ordi : P?????
Mot secret : P?????
Proposition du solveur : POPLITE
Saisir l'affichage de l'ordi : P***+++
Proposition du solveur : PRIRENT
Saisir l'affichage de l'ordi : P*+*+++
Proposition du solveur : PATINES
Saisir l'affichage de l'ordi : P+T+N+*
Proposition du solveur : PETUNIA
Saisir l'affichage de l'ordi : PETUNIA
```

On vous fournit une archive `materiel.zip` avec :

1. Un fichier `dico.txt` contenant les mots proposés, extrait directement du dépôt <https://framagit.org/JonathanMM/sutom> du projet officiel;
2. Un fichier `outils_sutom.py` à compléter qui constituera une bibliothèque d'outils, et un fichier de tests `test_outils_sutom.py` rassemblant les tests unitaires des fonctions de `outils_sutom.py`.
3. Un fichier `sutom_cli.py`, à compléter, qui proposera une simulation de partie interactive avec interface textuelle en ligne de commande (`cli` est un acronyme de `command line interface`).
4. Un fichier `sutom_gui.py`, à compléter, qui proposera une simulation de partie interactive avec interface graphique (`gui` est un acronyme de `graphic user interface`). L'interface graphique est construite à partir des fonctions du module `nsi_ui.py` fournie avec le manuel de **Première NSI Hachette** et décrite pages 52 et 53.
5. Un fichier `sutom_solver.py`, à compléter, qui proposera au choix :
 - a. la simulation d'une partie non interactive en interface textuelle où un solveur effectue les propositions en fonction des réponses de l'ordinateur;
 - b. l'exécution d'un assistant pour nous aider à résoudre le problème du jour proposé sur <https://sutom.nocle.fr/#>.

Un fichier de tests unitaires `test_sutom_solver.py` rassemble des tests unitaires pour les fonctions de `sutom_solver.py` et un test mesure de performance du solveur qu'il faudra compléter.

Voici le **cahier des charges** :

- ☞ Les scripts fournis dans l'archive `materiel.zip` doivent être complétés à partir des squelettes de code fournis et rassemblés dans une archive zip nommée `Eleve1_Eleve2_projet_sutom.zip` déposée dans l'espace prévu sur le Moodle du cours (accessible via l'ENT).
- ☞ Les questions marquées du logo  nécessitent une réponse écrite que vous saisirez dans un fichier `reponse.odt` ajouté à l'archive rendue. Pensez à faire apparaître les parties et les numéros des questions.
- ☞ Chaque fonction doit être documentée avec une `docstring`.
- ☞ Les parties les moins évidentes du code doivent être commentées de façon pertinente.
- ☞ Le code produit doit être confronté aux tests unitaires fournis dans les deux scripts de test. Les tests échoués doivent être mentionnés en commentaire dans les scripts de tests.

Objectif : *Thèmes du programme abordés*

1. *Types de données construits : tableaux, chaînes de caractères, dictionnaires;*
2. *Noyau d'un langage de programmation : boucles, tests, fonctions;*
3. *Interface Homme Machine : textuelle ou graphique;*
4. *Modularité : organisation d'un projet en unités fonctionnelles relativement indépendantes, séparation entre fonctions logiques et interface, regroupement des tests dans des fichiers.*
5. *Tester : tests unitaires de fonctions.*

2 Partie 1 : bibliothèque de fonctions outils

Objectif :

Dans cette partie on complète les codes de fonctions outils utilisées dans les autres scripts où sont développés les interfaces textuelle, graphique ou le solveur.

On illustre ainsi le principe de modularité et de séparation du fond et de la forme.

Extraite l'archive `materiel.zip` puis éditer le fichier `outils_sutom.py` dans un environnement de développement Python.

On donne ci-dessous la structure simplifiée du programme avec juste une description de ce que doit faire chaque fonction.

```
# coding: utf-8

#%% Import du module random
import random

#%% Définition des constantes globales
MAL_PLACE = '+'
ABSENT = '*'

#%% Définition des fonctions Outils

def charger_dico(dico_path, taille_secret):
    """
    Ouvre le dictionnaire de mots de chemin dico_path
    Précondition : un mot par ligne sans accent dans le fichier
    Extrait les mots de 6 lettres dans l'ordre
    Renvoie un tableau de mots
    """
```

```
def histogramme_dico(dico_path):
    """
    Renvoie un histogramme du nombre de mots par taille de mot
    dans le dictionnaire accessible par dico_path
    """

def tirage_mot(tab):
    """
    Sélectionne un mot au hasard dans le tableau de mots tab
    Précondition : tab est un tableau de mots (str) sans accents
    """

def rang_alpha(c):
    """
    Renvoie le rang (entre 0 et 25) du caractère c
    dans l'ordre alphabétique
    """

def histogramme(mot):
    """
    Renvoie l'histogramme des caractères d'un mot
    Précondition : mot contient uniquement des
    caractères en minuscule
    """

def verif_proposition(prop_joueur, secret):
    """
    Compare la proposition du joueur et le secret
    Renvoie un couple :
        un booléen déterminant si prop_joueur == secret
        une chaîne de caractères indiquant la distance
        entre prop_joueur et secret,
        avec les caractères MAL_PLACE, ABSENT ou le caractère trouvé
    """

def copie_tab(tab):
    """Renvoie une copie superficielle du tableau tab"""
```

Éditer également le fichier de tests unitaires `test_ouutils_sutom.py` dont la structure simplifiée est la suivante :

```
# coding: utf-8

#%% Import des fonctions à tester depuis le module outils_sutom

from outils_sutom import *

#%% Définitions des fonctions de test
```

```
def test_charger_dico():
    """Tests unitaires pour la fonction charger_dico"""

def test_histogramme_dico():
    """Tests unitaires pour la fonction histogramme_dico"""

def test_rang_alpha():
    """Tests unitaires pour la fonction rang_alpha"""

def test_histogramme():
    """Tests unitaires pour histogramme"""

def test_verif_proposition():
    """Tests unitaires pour verif_proposition"""

#%% Programme principal

# ne s'exécute pas si le script
# est importé dans un autre avec import test_outils_sutom
if __name__ == "__main__":
    test_charger_dico()
    test_histogramme_dico()
    test_rang_alpha()
    test_histogramme()
    test_verif_proposition()
```

1. Dans `outils_sutom.py`, lire attentivement la spécification de la fonction `charger_dico(dico_path, taille_secret)` puis la fonction de test unitaire `test_charger_dico()` dans `test_outils_sutom.py`. Compléter le corps de la fonction `charger_dico(dico_path, taille_secret)`. Vérifier que les tests unitaires sont passés en exécutant `test_outils_sutom.py`.
2. Dans `outils_sutom.py`, lire attentivement la spécification de la fonction `histogramme_dico(dico_path)` puis de la fonction de test unitaire `test_histogramme_dico()` dans `test_outils_sutom.py`. Compléter le corps de la fonction `histogramme_dico(dico_path)`. Vérifier que les tests unitaires sont passés en exécutant `test_outils_sutom.py`.
3. Dans `outils_sutom.py`, lire attentivement la spécification de la fonction `tirage_mot(tab)`. Compléter le corps de cette fonction. On ne fournit pas de tests unitaires.
4. Dans `outils_sutom.py`, lire attentivement la spécification de la fonction `rang_alpha(c)` puis la fonction de test unitaire `test_rang_alpha()` dans `test_outils_sutom.py`. Compléter le corps de la fonction `rang_alpha(c)`. Vérifier que les tests unitaires sont passés en exécutant `test_outils_sutom.py`.
5. Dans `outils_sutom.py`, lire attentivement la spécification de la fonction `histogramme(mot)` puis de la fonction de test unitaire `test_histogramme()` dans `test_outils_sutom.py`. Compléter le corps de la fonction `histogramme(mot)`. Vérifier que les tests unitaires sont passés en exécutant `test_outils_sutom.py`.

6. Dans `outils_sutom.py`, lire la spécification de la fonction `verif_proposition(prop_joueur, secret)` puis de la fonction de test unitaire `test_verif_proposition()` dans `test_outils_sutom.py`. Compléter le corps de la fonction `verif_proposition(prop_joueur, secret)`. Vérifier que les tests unitaires sont passés en exécutant `test_outils_sutom.py`.
7. Dans `outils_sutom.py`, lire attentivement la spécification de la fonction `copie_tab(tab)`. Compléter le corps de cette fonction. On ne fournit pas de tests unitaires.

3 Partie 2 : interface textuelle

Objectif :

Dans cette partie on complète le code d'une **interface textuelle** interactive permettant à l'utilisateur de jouer une partie de **sutom**.

L'ordinateur tire au hasard un mot dans le dictionnaire et le joueur essaie de le deviner. Les conventions d'affichage des lettres absentes du mot secret ou mal placées sont celles fixées dans le préambule.

Voici un exemple de partie :

```
Nouvelle partie (o/n) ? o
Taille du mot secret ? 6
Mot secret : P?????
Essai 1/6
Proposition du joueur : PASSER
Réponse de l'ordinateur : P*****
Essai 2/6
Proposition du joueur : PSAUME
Réponse de l'ordinateur : P*****
Essai 3/6
Proposition du joueur : PENSER
Réponse de l'ordinateur : PE*****
Essai 4/6
Proposition du joueur : PEINES
Réponse de l'ordinateur : PE***S
Essai 5/6
Proposition du joueur : PERDIS
Réponse de l'ordinateur : PE***S
Essai 6/6
Proposition du joueur : PETITS
Réponse de l'ordinateur : PETITS
Gagné en 6 essais
Le mot secret était PETITS
Nouvelle partie (o/n) ? n
```

Avant de répondre aux questions lire les pages 51 et 52 du manuel de **Première NSI Hachette** sur les interfaces textuelles.

Éditer le fichier `sutom_cli.py` dans un environnement de développement Python.

On donne ci-dessous la structure simplifiée du code avec juste la description de ce que fait chaque fonction. En plus des constantes notées en majuscules qui ne seront pas modifiées à l'exécution, on utilise une variable globale `jeu` de type **dictionnaire** pour stocker les différents paramètres. On peut modifier les valeurs des différentes clefs du dictionnaire `jeu` depuis l'intérieur de chaque fonction.

```
# coding: utf-8

# %% Import des fonctions outils
from outils_sutom import *

# %% Définition des constantes globales

DICO_PATH = "dico.txt"
ESSAI_MAX = 15
MAL_PLACE = '+'
ABSENT = '*'
DEBUG = True

# %% Définitions des fonctions pour l'interface en ligne de commande

def partie():
    """
    Exécution d'une partie :
        réinitialisation des paramètres
        boucle de partie
    """

def validation_joueur():
    """
    Saisie d'une réponse du joueur humain
    Appel de la fonction reponse_ordi avec la proposition du joueur
    """

def reponse_ordi(prop_joueur, secret):
    """
    Compare la proposition du joueur au secret
    Affiche la réponse de l'ordinateur
    Gère l'affichage de fin de partie
    Utilise la fonction verif_proposition
    """

def interface():
    """
    Interface textuelle avec boucle de jeu
    Utilise la fonction partie
    """

# %% Variable globale
# dictionnaire des paramètres du jeu
jeu = {"essai_max": ESSAI_MAX,
```



```
"essai": 1,
    "gagne": False,
    "verrou": True
}
# %% Programme principal
# ne s'exécute pas si le script
# est importé dans un autre avec import sutom_cli
if __name__ == "__main__":
    # boucle principale d'interface en ligne de commande
    interface()
```





1. On définit comme **constante** toute variable globale dont le nom est en majuscules et dont la valeur ne sera pas modifiée lors de l'exécution du programme.



Lister toutes les constantes disponibles lorsqu'on exécute `sutom_cli.py`.



Pensez à utiliser la fonction `dir()` pour afficher les variables accessibles dans le script.

2.  Décrire précisément le fonctionnement de la boucle d'interface textuelle exécutée par la fonction `interface()`. Préciser en particulier les conditions d'arrêt des différentes boucles imbriquées, le sens des interactions textuelles (entrée ou sortie) et les fonctions Python qui les rendent possibles.
3.  `verif_proposition(prop_joueur, secret)` est appelée dans `reponse_ordi(prop_joueur, secret)` alors que `verif_proposition` n'est pas définie dans `sutom_cli.py`. Comment est-ce possible?
4.  Que faut-il modifier dans `sutom_cli.py` si le chemin d'accès au fichier contenant le dictionnaire des mots proposés a changé?
5. Compléter le bloc de la fonction `validation_joueur()` en respectant sa spécification.
6. Compléter le bloc de la fonction `partie()` en respectant sa spécification.
7.  Question hors barème : À vous de jouer!

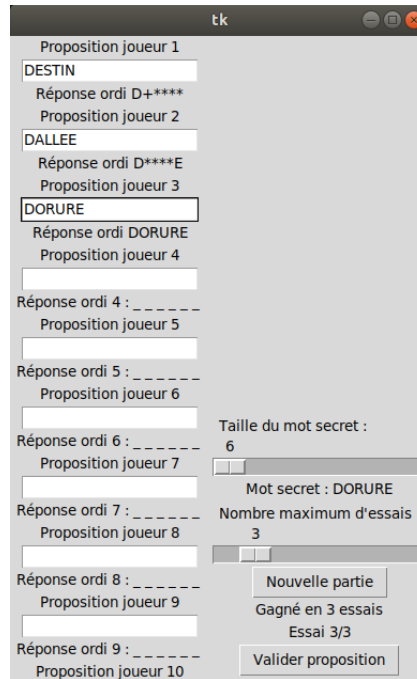
4 Partie 3 : interface graphique

Objectif :

Dans cette partie on complète le code d'une **interface graphique** interactive permettant à l'utilisateur de jouer une partie de **sutom**.

L'ordinateur tire au hasard un mot dans le dictionnaire et le joueur essaie de le deviner. Les conventions d'affichage des lettres absentes du mot secret ou mal placées sont celles fixées dans le préambule.

Voici un exemple de partie :



Avant de répondre aux questions lire les pages 51, 52 et 53 du manuel de **Première NSI Hachette** sur les interfaces graphiques.

Éditer le fichier `sutom_gui.py` dans un environnement de développement Python.

On donne ci-dessous la structure simplifiée du code avec juste la description de ce que fait chaque fonction. En plus des constantes notées en majuscules qui ne seront pas modifiées à l'exécution, on utilise une variable globale `jeu` de type **dictionnaire** pour stocker les différents paramètres. On peut modifier les valeurs des différentes clefs du dictionnaire `jeu` depuis l'intérieur de chaque fonction.

```
# coding: utf-8

# %% Import des fonctions d'interface graphique du module nsi_ui
from nsi_ui import *

# %% Import des fonctions outils du module outils_sutom
from outils_sutom import *

# %% Définition des constantes

DICO_PATH = "dico.txt"
ESSAI_MAX = 15
MAL_PLACE = '+'
ABSENT = '*'
DEBUG = True

# %% Définitions des fonctions pour l'interface graphique

def interface():
    """
    Définition et disposition en bloc des interacteurs de l'interface
    graphique
    """
```

```
"""



def partie():
    """
    Fonction de rappel pour le bouton jeu["bloc_jouer"] défini dans interface
    ()
    les champs d'étiquettes label de jeu["bloc_proposition"] et jeu["
        bloc_reponse"]
    les variables d'une partie : jeu["secret"], jeu["verrou"], jeu["gagne"]
    jeu["essai"] et jeu["essai_max"]
    """





def validation_joueur():
    """
    Fonction de rappel du bouton jeu["bloc_validation"]
    Valide le choix du joueur :
        vérification que la saisie est cohérente avec la longueur du secret
        affichage de la réponse de l'ordinateur
    Utilise la fonction reponse_ordi
    """

def reponse_ordi(prop_joueur, secret):
    """
    Vérifie la proposition du joueur avec verif_proposition(prop_joueur,
        secret)
    Affiche la réponse de l'ordinateur dans jeu["bloc_reponse"][essai - 1]
    Vérifie si le joueur a gagné ou atteint le nombre maximum
    et met à jour l'affichage de fin de partie dans jeu["bloc_gagne"]
    et jeu["bloc_annonce"], verrouille alors le jeu avec jeu["verrou"] = True
    Utilise la fonction verif_proposition
    """

# %% Variable globale
# dictionnaire des paramètres du jeu
jeu = {"essai_max": ESSAI_MAX,
       "essai": 1,
       "bloc_reponse": [],
       "bloc_proposition": [],
       "gagne": False,
       "verrou": True
       }

if __name__ == "__main__":
    interface() # Interface graphique
```


1.  Lister les différents types d'interacteurs utilisés dans cette interface graphique.
2.  Qu'est-ce qu'une fonction de rappel? Donner des exemples dans le code de `sutom_gui.py`.

3.  Combien de boucles `for` ou `while` le code `sutom_gui.py` contient-il?
4.  Pour le déroulement d'une partie, quelle fonction permet cependant une boucle d'interaction entre les interacteurs de l'interface?
5.  Pour les interfaces graphiques, on parle de *programmation guidée par les événements*. Expliquez cette expression.
Avez-vous déjà écrit des programmes guidés par les événements dans un contexte scolaire?
6.  Quel est le rôle de la valeur `jeu["verrou"]` ?
7. Compléter le code de la fonction `reponse_ordi(prop_joueur, secret)` pour gérer la fin de partie :
 - Affichage du message « Vous avez gagné en .. essais » dans l'interacteur `jeu["bloc_gagne"]` si le joueur a gagné.
 - Affichage du message « Vous avez perdu en .. essais » dans l'interacteur `jeu["bloc_gagne"]` si le joueur a perdu.
 - Affichage du mot secret dans l'interacteur `jeu["bloc_annonce"]`.
8. Écrire une fonction `interface2()` où la disposition des blocs est modifiée.


5 Partie 4 : solveur et assistant

Objectif :

Dans cette partie on complète le code d'un solveur de *sutom* qui peut fonctionner selon deux modes, selon la valeur d'une constante booléenne `ASSISTANCE`.

-  Si `ASSISTANCE` vaut `True` alors la fonction `interface_assistant()` est exécutée. Pour trouver le mot du jour, l'utilisateur saisit les réponses du serveur sur <https://sutom.nocle.fr/> et l'assistant lui propose un mot. Ci-dessous un exemple d'exécution pour trouver le mot du vendredi 28/01/2022.

```
Saisir l'affichage de l'ordi : S???????  
Mot secret : S???????  
Proposition du solveur : SCHERZOS  
Saisir l'affichage de l'ordi : S++++*  
Proposition du solveur : SARCELLE  
Saisir l'affichage de l'ordi : S++++*E  
Proposition du solveur : SILICOSE  
Saisir l'affichage de l'ordi : S++++*E  
Proposition du solveur : SUPPLICE
```

-  Si `ASSISTANCE` vaut `False` alors la fonction `interface_solveur()` est exécutée. L'utilisateur saisit une taille de mot secret puis l'ordinateur alterne entre les rôles de juge en ligne et de solveur. Le juge en ligne choisit un mot secret, le solveur propose une réponse etc ... comme dans `sutom_cli.py` mais en mode automatique.

```
Entrez la taille du mot secret : 7
Nouvelle partie (o/n) ? o
Nombre d'essais (1 -> 15) : 7
Mot secret : I??????
Proposition du solveur : INNOVAS
Réponse de l'ordinateur : I+****+
Proposition du solveur : IRONISA
Réponse de l'ordinateur : I*0****
Proposition du solveur : ISOSPIN
Réponse de l'ordinateur : ISOSPIN
Gagné en 3 essais
Le mot secret était ISOSPIN
```

Éditer les fichiers `sutom_solveur.py` et `sutom_solveur.py` dans un environnement de développement Python.

On donne ci-dessous la structure simplifiée du code avec juste la description de ce que fait chaque fonction. En plus des constantes notées en majuscules qui ne seront pas modifiées à l'exécution, on utilise une variable globale `jeu` de type **dictionnaire** pour stocker les différents paramètres. On peut modifier les valeurs des différentes clefs du dictionnaire `jeu` depuis l'intérieur de chaque fonction.

```
# coding: utf-8

# %% Import des fonctions outils du module outils_sutom
from outils_sutom import *
import random

# %% Définition des constantes

DICO_PATH = "dico.txt"
ESSAI_MAX = 15
MAL_PLACE = '+'
ABSENT = '*'
DEBUG = True
ASSISTANCE = False

# %% Fonction pour définir le solveur
def tirage_mot_sans_remise(tab):
    """
    Extrait un mot au hasard du tableau de mots tab
    """

def bien_place(caractere):
    """
    Détermine si un caractère lu dans la réponse de l'ordinateur est bien
    placé
    """
```

```
def compatible(mot, prop_solveur, rep_ordi):
    """
    Détermine si mot est compatible avec la dernière réponse
    de l'ordinateur
    """

def proposition_solveur(affichage):
    """
    Filtre les mots possibles dans le tableau jeu["mots_possibles"]
    Tire un mot au hasard parmi les mots possibles
    Affiche le mot sélectionné comme proposition du solveur
    Parametres:
        affichage (boolean) : détermine s'il y a des affichages avec print
    Retour: (None)
    """

# %% Fonctions pour définir l'interface en ligne de commandes

def reponse_ordi(prop_solveur, secret, affichage=True):
    """
    Compare la proposition du joueur au secret
    Affiche la réponse de l'ordinateur
    Gère l'affichage de fin de partie comme dans sutom_solveur.py
    """

def partie(affichage=True):
    """
    Exécution d'une partie :
        réinitialisation des paramètres
        boucle de partie
    """

# %% Interface

def interface_solveur():
    """
    Interface textuelle avec boucle de jeu
    Similaire à celle de sutom_cli.py
    """

# %% Assistant solveur



def interface_assistant():
    """Interface textuelle pour assistant de résolution de sutom"""
```

```
# %% Variable globale

# dictionnaire des paramètres du jeu
jeu = {"essai_max": ESSAI_MAX,
       "essai": 1,
       "mots_possibles": [],
       "propositions": [],
       "reponses": [],
       "gagne": False,
       "verrou": True
      }

# %% Programme principal

# ne s'exécute pas si le script
# est importé dans un autre avec import sutom_solveur
if __name__ == "__main__":
    taille_secret = int(input("Entrez la taille du mot secret : "))
    jeu["taille_secret"] = taille_secret
    jeu["mots"] = charger_dico(DICO_PATH, taille_secret)
    if ASSISTANCE:
        interface_assistant()
    else:
        interface_solveur()
```

1.  Lire attentivement le code fourni. En supposant que toutes les fonctions sont codées correctement, que se passe-t-il si on exécute :
 - a. `partie(affichage=True)`?
 - b. `partie(affichage=False)`?
2.  Lire attentivement le code de la fonction `proposition_solveur(affichage)` et compléter sa docstring pour décrire comment cette fonction sélectionne le mot proposé par le solveur.
3. Dans `sutom_solveur.py`, compléter les blocs des fonctions suivantes en respectant leur spécification :
 - a. `tirage_mot_sans_remise(tab)`
 - b. `bien_place(caractere)`. Vérifier en exécutant les tests unitaires `test_bien_place()` dans `test_sutom_solveur.py`.
 - c. `compatible(mot, prop_solveur, rep_ordi)`, cette fonction est le coeur du solveur, il doit exister de multiples façons plus ou moins efficaces de satisfaire sa spécification. Vérifier en exécutant les tests unitaires `test_compatible()` dans `test_sutom_solveur.py`.
 - d. `interface_assistant()`, l'interaction textuelle mise en place par cette fonction doit correspondre à l'exemple donné dans l'objectif de la partie 4. Vérifier en cherchant le mot du jour sur <https://sutom.nocle.fr/>.

4. Dans `test_sutom_solveur.py`, compléter le bloc de la fonction suivante pour satisfaire sa spécification : `test_performance_solveur(nb_parties, taille_secret)`.

Le but est de mesurer la performance du solveur développé dans `sutom_solveur.py` en calculant la fréquence de résolutions réussies et le nombre moyens d'essais sur un échantillon de `nb_parties`. On appellera `nb_parties` fois `partie(affichage=False)` qui renvoie un booléen (résolution réussie ou non) et le nombre d'essais.

Comparer les performances de votre solveur avec celles du corrigé qui résout les mots secrets en 4,3 essais en moyenne.